Eulerian Fluid Simulator

Liviu-George Bitiușcă

MSc Computer Animation & Visual Effects

Bournemouth University

August 2016

Contents

Abstract

This thesis describes the implementation of an Eulerian fluid simulator. A short overview and comparison of computational fluid dynamics (CFD) techniques for the entertainment industry is given, before proceeding to describe and detail the chosen technique for this project, from underlying mathematical concepts to high-level programming algorithms using C++ and OpenGL. The resulting simulator is analysed in terms of internal software architecture, performance, issues encountered during development and desired future improvements.

Keywords: Computational Fluid Dynamics, CFD, Fluid simulation, Grid-based, Eulerian solver, Navier-Stokes equations.

1 Introduction

1.1 Motivation

It is hard to understate the impact that Computational Fluid Dynamics (CFD) has within the greater domain of applied science and engineering -most people know about or at least expect physical phenomena being simulated in various fields of science-, but it's easy to overlook its importance and involvement in computer animation, where it usually has a supporting role. After all, the general public will always consider computer animation to be first and foremost a form of artistic expression rather than a demonstration of technical prowess and to some extent, that is completely justifiable: computer animation is science servicing art and not the other way around.

Fluid simulations are indispensable to the visual entertainment industry, as they allow creating physical phenomena (e.g. smoke, water, lava or any other flowing material) that would be impractical or downright impossible to produce through any other means. A great deal of effort has been made, especially in the last 20 years, in order to achieve ever-increasing levels of realism and results have often been impressive, to the point where simulation can be indistinguishable from reality. That being said, it is still a very much open area of research, even when only judging by recent SIGGRAPH presentations, with new methods and techniques being constantly invented or refined.

1.2 Project goals

From its earliest proposal stage, tackling this subject aimed first and foremost at a solid understanding of the underlying mathematical model employed in Eulerian fluid modelling, but also at solving a reasonably challenging software engineering problem. Considerable thought has been put not only in understanding how a fluid simulator's parts work together, but also into how to build and connect them in a manner that is as efficient and as simple as possible.

Speaking about simple, a word is in order about the desired extent of development (to be discussed in more detail in chapter 5): while having a feature-rich project is certainly something to aspire to, the most simple version of an Eulerian simulator proved very early on to be a difficult enough task for the author to justify setting that as the intended goal, with plans for future improvements and additions. There are no technical innovations brought forward, but rather tried-and-tested concepts, as described in research papers and books, are implemented. In that regard, the project can be thought of as an exercise in fully replicating already existing technology. The primary source of information during development was *"Fluid Simulation for Computer Graphics (Second*

Edition)" [Bridson 2015], from which the majority of important algorithms and general structure of the simulator have been inspired.

2 Related work

Computational Fluid Dynamics (CFD) is a branch of fluid mechanics (the other one being fluid statics) that employs numerical analysis and other mathematical algorithms to resolve fluid flows. Historically, its purpose lay with helping to solve various engineering problems, from biology to astrophysics to any other field that required understanding and visualization of how fluid changes under certain conditions and how it affected objects which it came in contact with. Historically, CFD techniques only pertained to engineering applications, primarily in the aerospace industry. However, rapid advancements in computer technology towards the end of the last century led to CFD eventually being used in computer animation.

Widely considered as the foremost figure in the development of CFD as a discipline and referenced by virtually all researchers of fluid dynamics for computer animation, is Francis H. Harlow, an American theoretical physicist who was the first one to use a computer to model fluid flow, at Los Angeles National Laboratory. As leader of the T3 dynamics group from 1957 to the late 1960s, he pioneered many techniques, some of them still in direct or indirect use today. Among them the PIC (Particle-In-Cell) method [Harlow et al. 1958], which three decades later would form the basis of FLIP (Fluid Implicit Particle) method [Brackbill and Rupel 1986], arguably the most widely used simulation method today and the MAC (Marker-And-Cell) method [Harlow et al. 1965], a simple, yet effective approach of tracking the fluid motion with marker particles – something that the current project uses as well.

These ideas made their way into the world of computer graphics particularly through the seminal work of Stam and Fiume [1993, 1995] for gaseous phenomena and Foster and Metaxas [1996] for liquid phenomena. The latter two's work in particular marked the first time complex liquid phenomena were efficiently animated using physically-based methods, with realism being provided through a finite difference approximation of the incompressible Navier-Stokes equations. From this moment on, preoccupation for CFD increased tremendously in the computer graphics community and by the end of the decade a large number of films started to utilize to their net advantage this new approach for creating believable dynamic effects.

Another significant contribution to the development of fluid simulation for computer animation around this time was that of Jos Stam, with his paper *"Stable Fluids"* [1999], in which an unconditionally stable solver of the Navier-Stokes equations was presented, based on a semi-Lagrangian advection scheme (to be detailed in the next chapter) and implicit differentiation of viscosity, as opposed to the explicit and thus prone to numerical instability, method used earlier by Foster and Metaxas. Stam's innovations made a reasonable compromise between speed and accuracy, essentially proving that fluid simulations could be efficient enough to run even on consumer-grade hardware.

The early 2000s brought such accelerated development of fluid dynamics in computer graphics that films (and later, games) which didn't use this new technology quickly became the exception rather than the rule. Interactive fluid solvers made their way to commercial 3D applications (e.g. Maya Fluid Effects, a solver for phenomena such as smoke, fire, clouds or explosions was included in the software starting with Maya version 4.5 in 2003), thus making it easier than ever before for the technology to expand into all areas of computer graphics. A close inspection of the nature of special effects used by films in the early 2000s leads to the conclusion that it was precisely at this time that practical dynamic effects were irrevocably superseded by their digital counterparts.

A pivotal innovation in the new millennium was the introduction of the level set -a type of implicit surface that superseded the height field in accurately describing a liquid free surface [Foster and Fedkiw 2001; Osher and Fedkiw 2001], and a modification of it, the *hybrid particle* level set [Enright et al. 2002] -a mix between the original particle based method and the initial level set concept. Also, the late 2000s saw FLIP becoming an increasingly prevalent method, due to its reduced numerical dissipation when compared to pure Eulerian methods, as well as due to its good preservation of fine detail around the surface of the fluid. All high-end fluid solvers in active development today (Bifrost (ex-Naiad), Realflow, Houdini etc.) currently use FLIP in one form or another. Notable research work on FLIP has been done by Zhu and Bridson [2005] and very recently by Ferstl et al. [2016].

Apart from grid-based (Eulerian) methods previously described (technically some of them are hybrid methods, but they still rely on a spatial grid), there are also pure particle-based (Lagrangian) methods. This family of methods does not rely on spatial discretization, rather stores all the required information to perform the simulation on the particles. Their main advantage over grid based methods is the unbounded domain of simulation and decreased memory consumption, but coming at the cost of increased computation time (by comparison with FLIP, SPH requires 7 to 20 time steps or more each frame to stabilize [SideFX Software 2016]) and inability to accurately describe large bodies of water. By far the most popular of particle-based method is Smoothed Particle Hydrodynamics, or SPH, introduced in 1977 by Gingold and Monaghan for astrophysics applications and proposed as a viable solution for real-time and interactive applications by Müller et al. [2003]. The innovation of this system is that particles have a “smoothing length” over which properties are interpolated by a kernel function. This means properties of a particle (like pressure or temperature) can influence and be influenced by those of neighbouring particles.

3 Governing equations

Most fluid flow of interest to computer animation is governed by the incompressible **Navier-Stokes equations**, which are a set of partial differential equations:

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} + \frac{1}{\rho} \nabla p = \vec{g} + \nu \nabla \cdot \nabla \vec{u} \,,$$
$$\nabla \cdot \vec{u} = 0 \,.$$

where $\vec{u}$ is the **velocity** of the fluid $\vec{u} = (u, v, w)$, the Greek letter ρ is the **density** of the fluid, p is the **pressure** that the fluid exerts on anything, $\vec{g}$ is the sum of all **external forces** acting on the fluid (e.g. gravity) and the Greek letter ν is the **kinematic viscosity**, a measure of how much the fluid resists deformation or how easy it conforms to its container.
Now is a good time to mention that for many types of fluid, the viscosity term is not needed, so it is simply dropped from the equation. Dense fluids like honey or lava are highly dependent on viscosity, but for the majority of situations, viscosity does not play an important role. Water or air, for instance, have hardly any viscosity and are good examples of **inviscid** fluids (even though in real life they are not ideal inviscid fluids, their viscosity is low enough to be negligible in computer animation). In any case, due to the numerical methods used in approximating the solution to the fluid equations, some viscosity is unavoidably added into the simulation even when the viscosity term is dropped. The Navier-Stokes equations which are missing the viscosity term are called the **Euler equations** and they are in fact the ones that used in the current project:

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} + \frac{1}{\rho} \nabla p = \vec{g} \,, \tag{1.1}$$
$$\nabla \cdot \vec{u} = 0 \,. \tag{1.2}$$

3.1 The Momentum Equation

Equation (1.1) is called the **momentum equation** and in the world of fluid dynamics it is the equivalent of Newton's Second Law of motion: $\vec{F} = m\vec{a}$. What it describes then is how the fluid accelerates due to forces (internal and external) acting on it. It might not be immediately apparent, but the terms in the Euler equations exactly mirror those in Newton's Second Law, as it's about to be made a little more clear below.

To help with intuition, a particle-driven fluid simulation is considered, where each particle represents a small blob of fluid with a mass *m*, a volume *V* and a velocity $\vec{u}$. As

Newton's Second Law states, the force $\vec{F}$ on the particle is equal to its mass m times acceleration $\vec{a}$:

$$\vec{F} = m\vec{a} \tag{1.3}$$

Acceleration is not known, but velocity on the particle is indeed known, and since acceleration is the derivative of velocity, the equation can be rewritten as:

$$\vec{F} = m\frac{D\vec{u}}{Dt} \tag{1.4}$$

where $\frac{D\vec{u}}{Dt}$ is the notation used for the **Material Derivative** (to be dissected soon), but which for now can be safely thought of as just a standard derivative. The next step is to consider what forces are acting on the particle. Naturally the first one to be considered is the force of gravity (important to remember this is the *force* of gravity, not to be confused with simple *acceleration* due to gravity, which does not involve mass):

$$\vec{F}_{gravity} = m\vec{g}$$

Gravity is thus an external force, acting in equal amounts over each and every particle of the fluid and requires no further explanation. The somewhat more complex forces are those generated internally by the fluid on itself or, in other words, the forces which particles exert on their surrounding particles. There are two such forces: viscosity and pressure, but since viscosity has been previously dropped from the equation, only pressure is of importance. Thinking about it conceptually, fluid flows from high pressure areas to low pressure areas. To determine the force of pressure at a specific particle position, given that pressure is a spatial set of scalar values (a.k.a. a scalar field), the gradient of pressure, ∇p must be taken, resulting in a vector pointing in the direction of the "steepest ascent", which is then negated, $-\nabla p$, to point *away* from the region with high-pressure towards the region with low-pressure. To be perfectly clear, ∇p is just a collection of a scalar function's partial derivatives into a vector:

$$-\nabla p = -\begin{bmatrix} \partial / \partial x \\ \partial / \partial y \\ \partial / \partial z \end{bmatrix} \cdot p = -\begin{bmatrix} \partial p / \partial x \\ \partial p / \partial y \\ \partial p / \partial z \end{bmatrix}$$

To get the pressure force, the vector above then needs to be integrated over the volume of the particle blob, but as a simple approximation it can just be multiplied by *V* instead:

$$\vec{F}_{pressure} = -V\nabla p$$

At this stage it is still a bit unclear what the pressure's role will be in the actual simulation, but understanding the core idea that high pressure areas push fluid away according to the direction pointed to by the negative gradient of pressure is extremely important (more on this is section 4.5). Now, if equation (1.4) is rewritten with the new pieces of information, it will look like this:

$$m\vec{g} - V\nabla p = m\frac{D\vec{u}}{Dt}$$

Rearranging slightly, to have the acceleration term (the material derivative) on the left side, gives as the equation of motion for a blob of fluid to be:

$$m\frac{D\vec{u}}{Dt} = -V\nabla p + m\vec{g}$$

If calculation of a fluid is made, with the equation above, using a small finite number of particles, there are going to be approximation errors – values from sampled particles cannot fully account for values of unsampled ones. The solution is then to use a very large number of particles, still finite but approaching infinity, to describe the fluid. This is what is known as a **continuum model** and, according to Bridson [2015]: "has been shown experimentally to be in extraordinarily close agreement with reality in a vast range of scenarios". The downside of it is that as the number of particles in the system tends to infinity, the mass *m* and volume *V* of each particle tend to 0, thus making the equations of motion meaningless. In order to avoid that, before taking the limit as the number of particles goes to infinity, the momentum equation is divided by the volume:

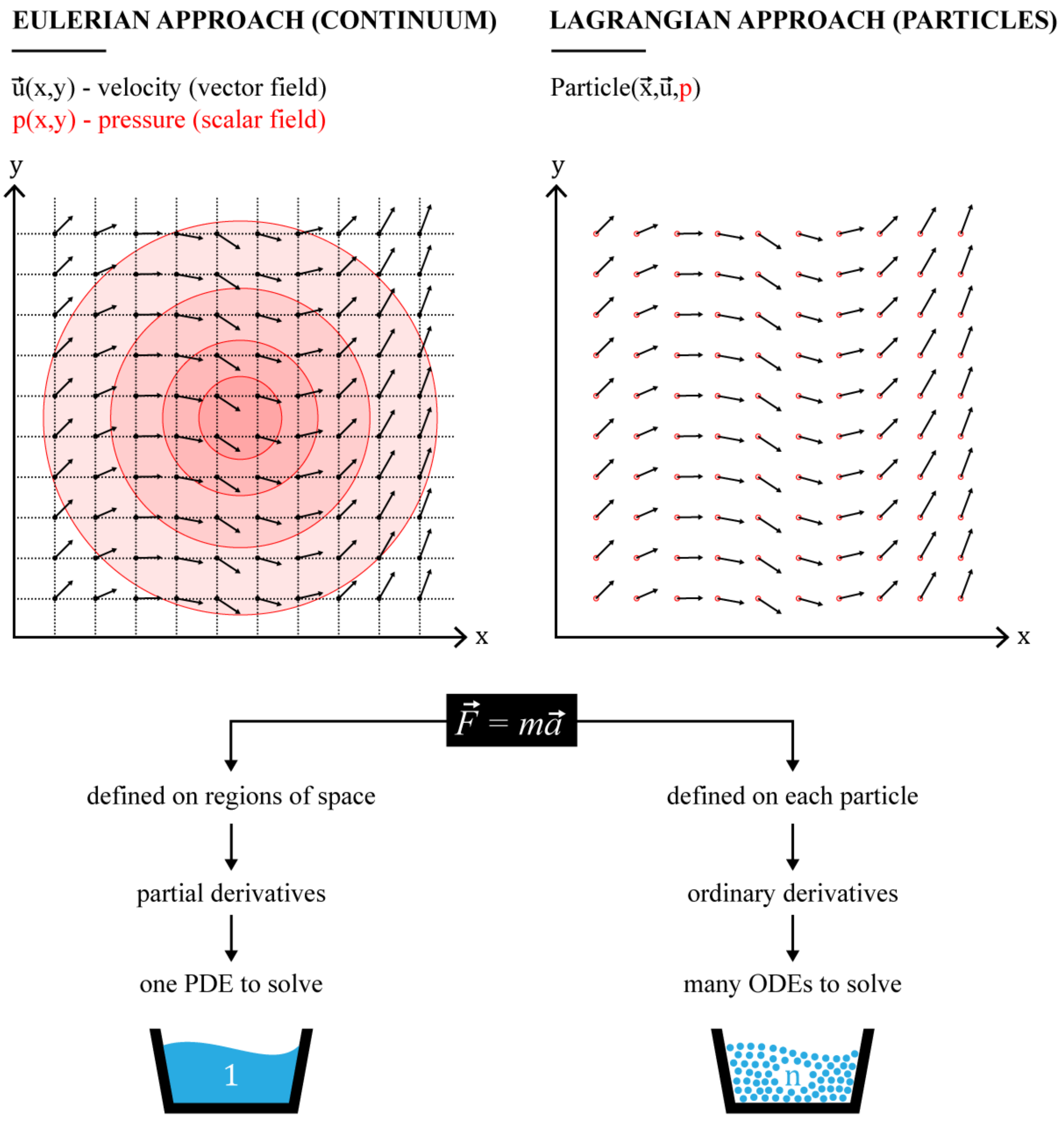
$$\frac{m}{V}\frac{D\vec{u}}{Dt} = -\nabla p + \frac{m}{V}\vec{g}$$

Knowing that mass/volume equals density, *m/V* can be replaced with ρ (rho) to give:

$$\rho\frac{D\vec{u}}{Dt} = -\nabla p + \rho\vec{g}$$

Finally, the material derivative can be isolated by dividing by ρ, to yield the final form of the momentum equation, one that will help with solving it numerically:

$$\frac{D\vec{u}}{Dt} = -\frac{1}{\rho}\nabla p + \vec{g} \tag{1.5}$$



EULERIAN APPROACH (CONTINUUM)
u⃗(x,y) - velocity (vector field)
p(x,y) - pressure (scalar field)
y
x
LAGRANGIAN APPROACH (PARTICLES)
Particle(x⃗,u⃗,p)
y
x
$\vec{F} = m\vec{a}$
defined on regions of space
partial derivatives
one PDE to solve
1
defined on each particle
ordinary derivatives
many ODEs to solve
n

Figure 1 Eulerian and Lagrangian viewpoints

3.2 The Material Derivative

So far, the acceleration of the particle has been treated as just a regular derivative of velocity. In a continuum model (e.g. fluid or deformable solid) however, there is more than one way to track motion (Figure 1).

The first one is the Lagrangian method (named after the Italian mathematician Joseph-Louis Lagrange) and it's what drives particle systems: points in space have a position $\vec{x}$ and a velocity $\vec{u}$. The body of the fluid is represented with many such particles and, knowing the active forces on them, allows the fluid to be advanced through time. This is the approach that Smoothed Particle Hydrodynamics, for instance, takes.

The other approach is the Eulerian one (named after Swiss mathematician Leonhard Euler), which instead looks at fixed locations in space and measures the change in the measurement of values (velocity, density, temperature or any other value) at those locations to determine how the fluid flows through the analyzed region. While not particularly intuitive, this approach has the advantage of allowing for easier approximation of spatial derivatives (e.g. pressure, temperature).

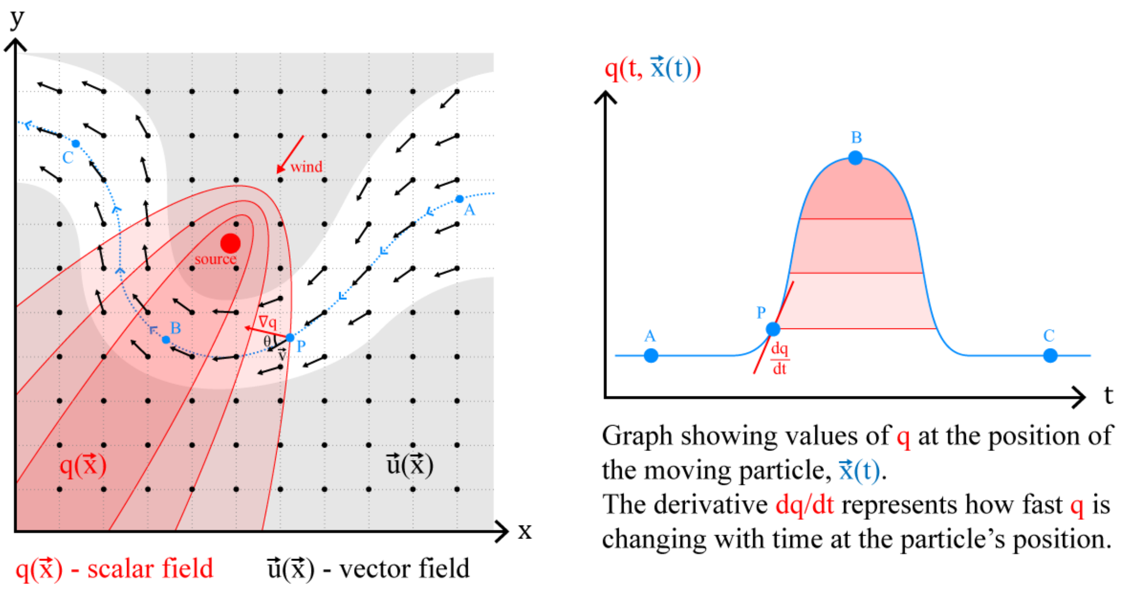

y
x
wind
source
A
B
C
P
∇q
θ
v
q(x⃗)
u⃗(x⃗)
q(x⃗) - scalar field
u⃗(x⃗) - vector field
q(t, x⃗(t))
t
B
A
P
C
dq/dt
Graph showing values of q at the position of the moving particle, x⃗(t).
The derivative dq/dt represents how fast q is changing with time at the particle's position.

Figure 2 The Material Derivative

What connects the two approaches is the Material Derivative, as it accounts for changes from both Lagrangian and Eulerian viewpoints. Figure 2 shows on the left two overlapping fields: a velocity field $\vec{u}(\vec{x})$ and a scalar field $q(\vec{x})$. A particle's trajectory is traced by the blue line. An analogy could be made that the particle moves through a river described by $\vec{u}$ and that, between points A and C, it goes through a smoke cloud (scattered from a nearby source) described by q. The smoke cloud is a scalar field, from which a scalar measurement -smoke density for instance- can be obtained at different points in space.

The problem at hand is just how fast q is changing not for a fixed point in space, $\vec{x}$, but rather for the particle whose position is given by $\vec{x}(t)$ as a function of time. Position P at time t, given by $\vec{x}(t)$ is on the particle's path and the particle has velocity $\vec{u}(P)$ at that point. The maximum rate at which q could change from P is given by its gradient, ∇q, which is a vector pointing from the point P towards the region of q with the highest increase in value (or the "steepest ascent"). Therefore how fast q is changing is determined mainly by how much the velocity vector points in the same direction as the gradient vector and of course by their magnitude (a very small velocity, even in the same direction as the gradient, would still mean a slow change). So the easiest way to see how much two vectors point in the same direction is to simply take their dot product: $\vec{u}\cdot\nabla q$ (also equivalent to $|\vec{u}||\nabla q|\cos\theta$). This is half of the problem solved. Next, changes in q that are Eulerian (so not a function of particles), like density variation due to wind direction over time, $\partial q/\partial t$, are also accounted for. The equation below is thus obtained for the derivative of q at a moving particle's position:

$$\frac{Dq}{Dt}=\frac{\partial q}{\partial t}+\vec{u}\cdot\nabla q \tag{1.6}$$

This is the Material Derivative and just to be thorough, below it is written in its expanded form:

$$\frac{Dq}{Dt}=\frac{\partial q}{\partial t}+u\frac{\partial q}{\partial x}+v\frac{\partial q}{\partial y}+w\frac{\partial q}{\partial z} \tag{1.7}$$

The last checked concept in this section is how the Material Derivative is applied to vector functions, especially to velocity, which advects itself (advection is discussed in Chapter 4). The idea is similar: combine Eulerian and Lagrangian parts to get a compound derivative for every component of the vector:

$$\frac{D\vec{u}}{Dt}=\frac{\partial \vec{u}}{\partial t}+\vec{u}\cdot\nabla\vec{u}=\begin{bmatrix} Du/Dt \\ Dv/Dt \\ Dw/Dt \end{bmatrix}=\begin{bmatrix} \partial u/\partial t+\vec{u}\cdot\nabla u \\ \partial v/\partial t+\vec{u}\cdot\nabla v \\ \partial w/\partial t+\vec{u}\cdot\nabla w \end{bmatrix} \tag{1.8}$$

3.3 Fluid incompressibility

In the real world, all fluids manifest *some* level of compressibility, but for the most part that goes unnoticed to the naked eye. No one can "see" sound waves for instance, which are perturbations of air volume. Therefore as far as computer animation is concerned, fluid compressibility can be ignored altogether and all fluid modeled **incompressible**. When a fluid is incompressible, it does not change its volume and from a mathematical standpoint that means that there is neither inflow nor outflow around the surface of the fluid: it simply maintains whatever volume Ω it already has. To satisfy this, the normal component of velocity around the fluid surface $\partial\Omega$ has to be zero:

$$\frac{d}{dt}volume(\Omega) = \iint_{\partial\Omega} \vec{u} \cdot \hat{n} = 0$$

Using the divergence theorem, this integral can be changed to a volume integral:

$$\frac{d}{dt}volume(\Omega) = \iiint_{\Omega} \nabla \cdot \vec{u} = 0$$

which must be true for any Ω part of the fluid. And the only continuous function that integrates to zero independent of the region of integration is zero itself, therefore the integrand needs to be zero everywhere [Bridson 2015]:

$$\nabla \cdot \vec{u} = 0$$

This is known as the **incompressibility condition** and it is the other part of the incompressible Navier-Stokes equations (the first one being the momentum equation). As will later be shown, in order to practically satisfy this condition, the velocity field of the fluid needs to be divergence-free and to enforce that, the pressure from the momentum equation is used.

3.4 Boundary conditions

Fluid must first of all not be allowed to go through the solid walls of its container and second of all (most evident in the case of liquids), a boundary needs to exist between it and its surrounding environment – this is what is known as the fluid's **free surface**. For a static solid boundary, velocity needs to be set to zero in the component *perpendicular* to the surface of the solid ($\hat{n}$ is the normal to the solid boundary):

$$\vec{u} \cdot \hat{n} = 0$$

Tangential velocity along the surface of the solid, on the other hand, can either be zero for viscous fluids (known as the **no-slip** condition) or left unaltered (the **no-stick** condition) for inviscid fluids. In the current project the latter condition is used.

Finally, the free surface condition is handled by setting the pressure outside the fluid to zero and not controlling the velocity in any way. In the implementation section it will be shown how velocities from the fluid are extrapolated in the free space, but that is only to aid with correctly interpolating velocities at the boundary of the fluid.

4 Solving the equations

Having the mathematical description of fluid motion in place, the next step in creating a fluid simulation is to solve the equations, or to be more precise to approximate their solutions in a manner that is as accurate as possible. This chapter details on how to do this.

4.1 Splitting the fluid equations for numerical simulation

Splitting is a technique where components of a more complicated equation are solved in turn, their effects then added up to form the complete solution. Apart from simplifying the solving process, splitting allows for use of different numerical methods for different terms, depending on what is better suited (for instance, gravity could very well use a forward Euler scheme, since it is a constant force, whereas advection would most likely require a more precise method, like Runge-Kutta 2nd order or higher). Therefore instead of solving the Euler equations in one step, they are split into parts:

$$\frac{D\vec{u}}{Dt} \text{ (advection)}$$

$$\frac{\partial \vec{u}}{\partial t} = \vec{g} \text{ (body forces)}$$

$$\frac{\partial \vec{u}}{\partial t} + \frac{1}{\rho}\nabla p = 0, \text{ such that } \nabla \cdot \vec{u} = 0 \text{ (pressure/incompressibility)}$$

With splitting, the high-level fluid algorithm becomes:

1. Start with an initial divergence-free velocity field $\vec{u}^0$
2. For timestep n = 0,1,2,3….n

   2.1 Determine a good timestep $\vartriangle t$ to go from time t_n to time t_{n+1}

   2.2 Advect the velocity field $\vec{u}^0$ to get $\vec{u}^A$

   2.3 Apply external forces $\vec{g}$ to $\vec{u}^A$ to get $\vec{u}^B$

   2.4 Make $\vec{u}^B$ divergence free and enforce incompressibility

Figure 3 Basic fluid algorithm

4.2 The MAC grid

Before going ahead with actually solving the equations to bring the fluid to life, various fluid properties and quantities need to be spatially discretized. At a basic level, velocity and pressure need to be mapped onto 3D space (other values can be discretized as well, but this project only concerns itself with the above two) and the way to do that is by means of a three-dimensional grid.

Introduced by Harlow and Welch [1965], the Marker-And-Cell (MAC) method for solving incompressible fluid flow involves creating a spatial grid onto which variables are stored at different locations (a **staggered** arrangement). The reason for the staggered arrangement is not immediately obvious, but as it will revealed, it greatly simplifies calculating pressure to enforce incompressibility. Figure 4 illustrates a two-dimensional MAC grid. As can be seen, at the center of each cell (i, j) a pressure $p_{i,j}$ is stored. Velocity, on the other hand, is not stored at the cell center, but rather split into components which are then stored at the centers of cell faces, with every face being shared by two neighbouring cells. The horizontal *u*-component is stored at the centers of vertical faces (red) and the vertical *v*-component is stored at the centers of the horizontal faces (green). The same thing happens in three dimensions (Figure 5).

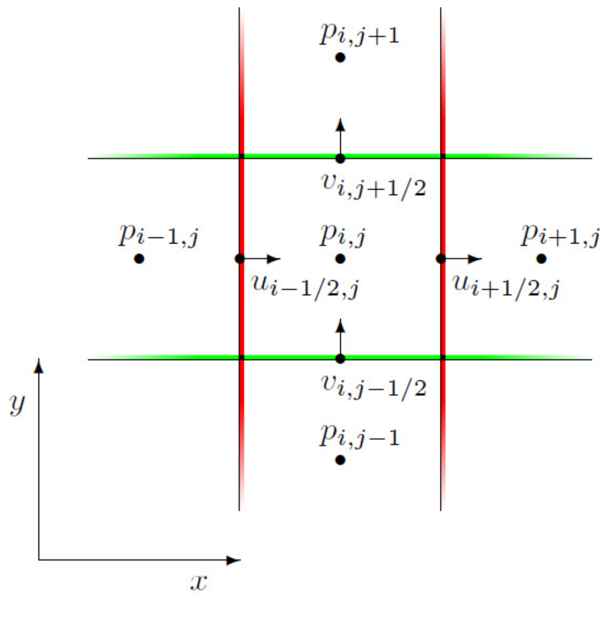


Figure 4 The 2D staggered MAC grid. Original image by Bridson (2015).

The reason for this staggered arrangement of pressure and velocity is explained in detail by [Bridson 2015], but the main idea is that it allows for accurate central differences when calculating spatial derivatives, such as the derivative of *u* at point *i*:

$$\left(\frac{\partial u}{\partial x}\right)_i \approx \frac{u_{i+1/2} - u_{i-1/2}}{\triangle x} \tag{2.1}$$

The strange half indices simply indicate that the position of the velocity is not at the cell centers, but rather halfway between cell centers. In actual code these velocities will of course be referred to by integer indices, but it makes more intuitive sense of using half indices when describing the algorithms and formulas.

A downside of using the staggered arrangement, however, is the added complexity when interpolating velocity. Because no vectors of velocity are actually stored, three interpolations are required (one per each component) every time the actual velocity vector is requested at any known or arbitrary point inside the grid. For arbitrary points, a bilinear (in 2D) or trilinear (3D) interpolation is always required, but at grid points (where we store pressure) and cell face centers (where we store the components of velocity themselves) averaging is enough:

$$\vec{u}_{i,j,k} = \left(\frac{u_{i-1/2,j,k} + u_{i+1/2,j,k}}{2}, \frac{v_{i,j-1/2,k} + v_{i,j+1/2,k}}{2}, \frac{w_{i,j,k-1/2} + w_{i,j,k+1/2}}{2} \right) \tag{2.2}$$

$$\vec{u}_{i+1/2,j,k} = \left(u_{i+1/2,j,k}, \frac{\begin{matrix} v_{i,j-1/2,k} + v_{i,j+1/2,k} \\ + v_{i+1,j-1/2,k} + v_{i+1,j+1/2,k} \end{matrix}}{4}, \frac{\begin{matrix} w_{i,j,k-1/2} + w_{i,j,k+1/2} \\ + w_{i+1,j,k-1/2} + w_{i+1,j,k+1/2} \end{matrix}}{4} \right) \tag{2.3}$$

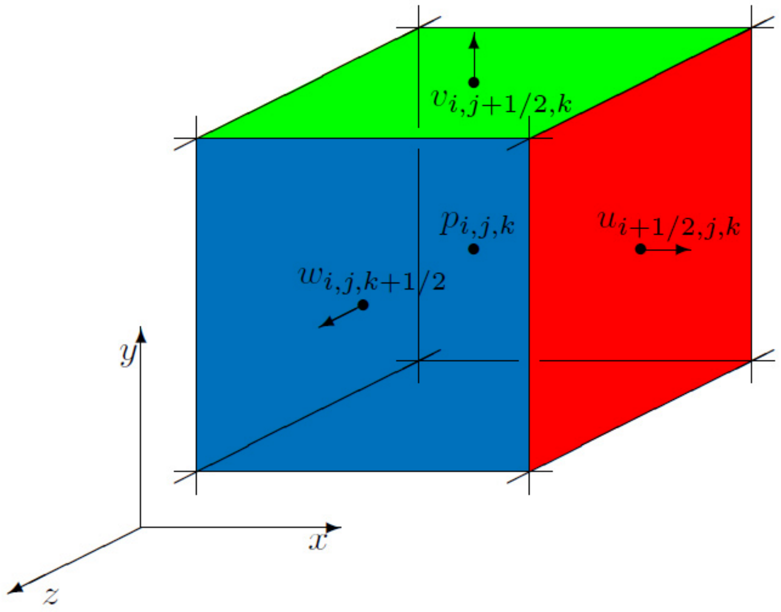


Figure 5 Cell of a 3D staggered MAC grid. Original image by Bridson (2015).

4.3 Advection

Advection describes how particles (or blobs) of fluid move with the velocity field $\vec{u}$. The **advection equation** states that quantities being advected do not change in the Lagrangian viewpoint, but simply move around:

$$\frac{Dq}{Dt} = \frac{\partial q}{\partial t} + \vec{u} \cdot \nabla q = 0 \tag{2.4}$$

For example, if particles each had a value for temperature, that value would not change on the particles as they were being moved around by the velocity field. Section 1.3.1 in *"Fluid Simulation for Computer Graphics"* [Bridson 2015] analyzes a detailed example.
The approach taken to solve the advection is the **Semi-Lagrangian** method introduced by Stam [1999], which is both easy to understand, fairly easy to implement and unconditionally stable. The idea behind semi-Lagrangian advection is that instead of using forward integration for the time derivative $\partial q / \partial t$ and an accurate central difference for the spatial derivative $\vec{u} \cdot \nabla q$, a backwards particle trace is performed from the point of interest.

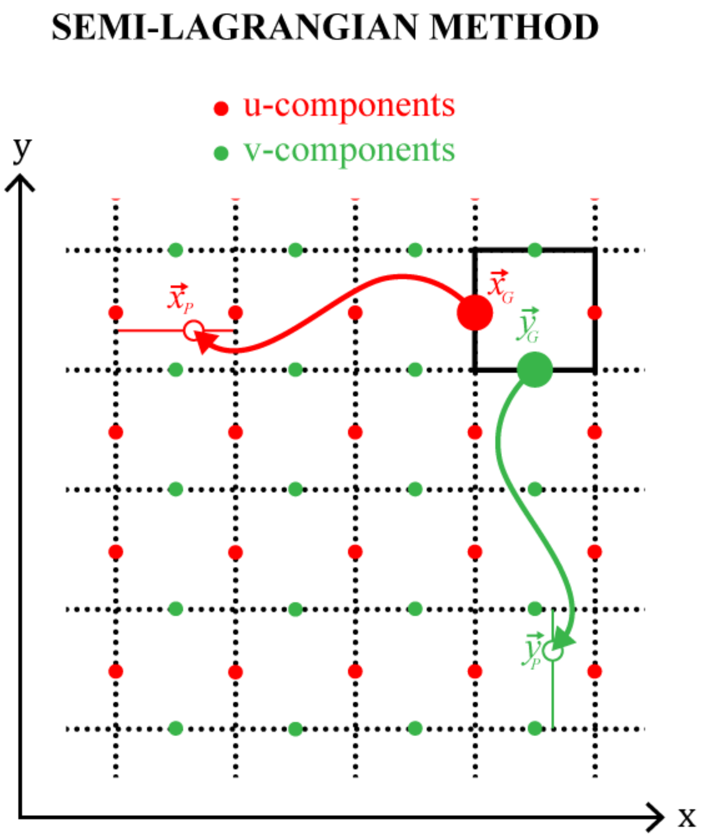


Figure 6 The Semi-Lagrangian method

Looking at a practical example, Figure 6 depicts the staggered MAC grid with u and v velocity components stored at cell faces. To find out the value of u at position $\vec{x}_G$ at the new timestep, an imaginary particle (hence the "semi" in "semi-Lagrangian") is traced back one timestep using the reversed velocity field to its old position $\vec{x}_P$. There, an interpolation between the two closest u-components is performed to find the old u value, which is then directly assigned to $\vec{x}_G$. A similar step is performed to find the value of v at $\vec{y}_G$ and, for a three-dimensional grid, for w at $\vec{z}_G$ as well. The advection algorithm for velocity in a 3D-scenario is outlined below:

```
For every cell (i, j, k):
    Loop over all velocity components (u, v, w):
        # routine for the u-component is outlined below
        • Perform interpolation at the location $\vec{x}_G$ where the velocity component $u_G$
          is stored to find the full velocity vector.
        • Reverse the velocity vector.
        • Integrate one timestep (example is forward Euler, but Runge-Kutta 2nd
          order or higher is recommended to get accurate results):
          $\vec{x}_P = \vec{x}_G + \vec{u}_G \Delta t$
        • At $\vec{x}_P$, interpolate the velocity component $u_P$.
        • Use the old value at the old position as the new value at the new position:
          $u_G = u_P$
```

Figure 7 Advection algorithm

4.4 Velocity extrapolation and boundary conditions

Given the way advection is performed, it is entirely possible for the imaginary particle to be traced back to a position that lies outside the fluid boundaries, for instance inside a solid wall or in the open air. In order for the velocity interpolation to be performed correctly there, something needs to be done to extend the known fluid velocities into areas where it is unknown. This is called **velocity extrapolation**.

The case where the particle ends up inside a solid is simpler. As per the reasoning described in section 3.4, tangential velocities inside solids need to be set to something that would not hinder movement of the fluid *along* the solid walls. The natural choice is to simply mirror the velocity values inside the fluid, as shown in Figure 8. This means no change in tangential velocity when interpolation at the boundary is performed (interpolation between two identical values returns the same value).

Extrapolation of velocity from the fluid to the surrounding air is a bit more involved, however will not be described here (a lot of algorithms for extrapolation can be found with a quick internet search), but it basically involves averaging velocities starting from the surface of the fluid outwards in the surrounding air.

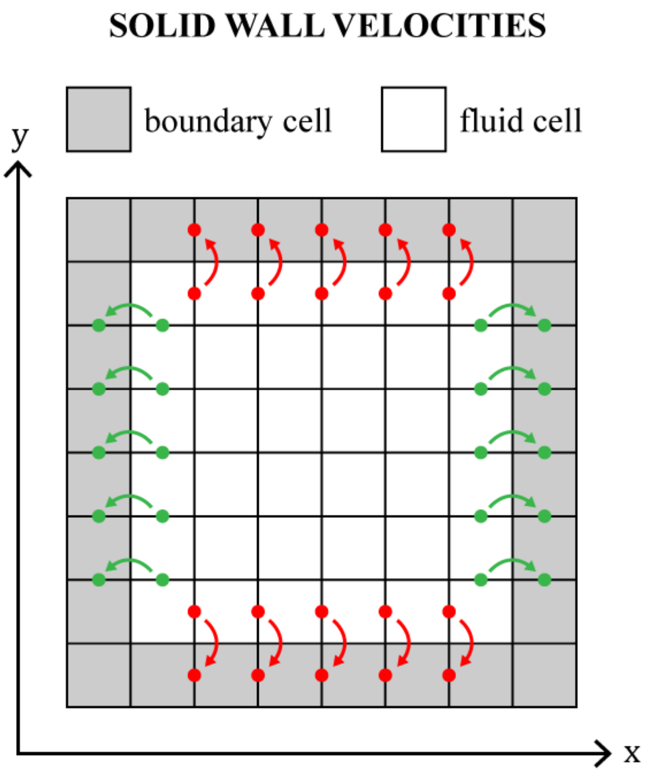


Figure 8 Extrapolating fluid velocities to solid walls (the **no-stick** condition)

4.5 The pressure gradient

The velocity field obtained after advection and addition of any external forces is not divergence-free, thus would not preserve fluid volume. To fix that, a new force needs to be added into the field, one that would force it to become free of divergence, leading to fluid incompressibility and simultaneously enforcing boundary conditions.

As described in section 3.1, high pressure areas push fluid away according to the direction pointed to by the negative gradient of pressure. So for the actual update of velocity at time *n+1*, as postulated by the momentum equation, the gradient of pressure needs to be subtracted (can also be thought of as the *negative* gradient of pressure being *added* if force addition makes more sense to the reader) from the intermediate velocity field $\vec{u}$ generated by the advection step:

$$\vec{u}^{n+1} = \vec{u} - \Delta t \frac{1}{\rho} \nabla p \tag{2.5}$$

with the resulting velocity field satisfying the incompressibility condition

$$\nabla \cdot \vec{u}^{n+1} = 0 \tag{2.6}$$

and also solid wall boundary conditions $\vec{u}^{n+1} \cdot \hat{n} = 0$ and free surface condition that p = 0.

Now it becomes clear why a staggered arrangement of variables is used. When a component $\frac{\partial p}{\partial x}, \frac{\partial p}{\partial y}$ or $\frac{\partial p}{\partial z}$ of the gradient of pressure ∇p needs to be subtracted from the *u*, *v* or *w*-component of velocity $\vec{u}$, there are two pressure values lined up perfectly on either side of the velocity component. Therefore equation (2.5) can be approximated using central differences for ∇p as:

$$\begin{aligned}
u^{n+1}_{i+1/2,j,k} &= u_{i+1/2,j,k} - \Delta t \frac{1}{\rho} \frac{p_{i+1,j,k} - p_{i,j,k}}{\Delta x} \\
v^{n+1}_{i,j+1/2,k} &= v_{i,j+1/2,k} - \Delta t \frac{1}{\rho} \frac{p_{i,j+1,k} - p_{i,j,k}}{\Delta x} \\
w^{n+1}_{i,j,k+1/2} &= w_{i,j,k+1/2} - \Delta t \frac{1}{\rho} \frac{p_{i,j,k+1} - p_{i,j,k}}{\Delta x}
\end{aligned} \tag{2.7}$$

Not all the velocities in the grid need to be updated with the pressure gradient though. Air regions which do not border any fluid, for example, do not need to be updated, since preserving the air's volume is not important. Velocity at solid walls also doesn't need to be updated with the pressure gradient, since it is set directly as per the boundary conditions (zero for perpendicular flow, free for tangential flow). The only velocities that need to be updated with the pressure gradient are those inside the fluid and those at the fluid's free surface (which border air cells). Refer to Figure 9 for clarification.

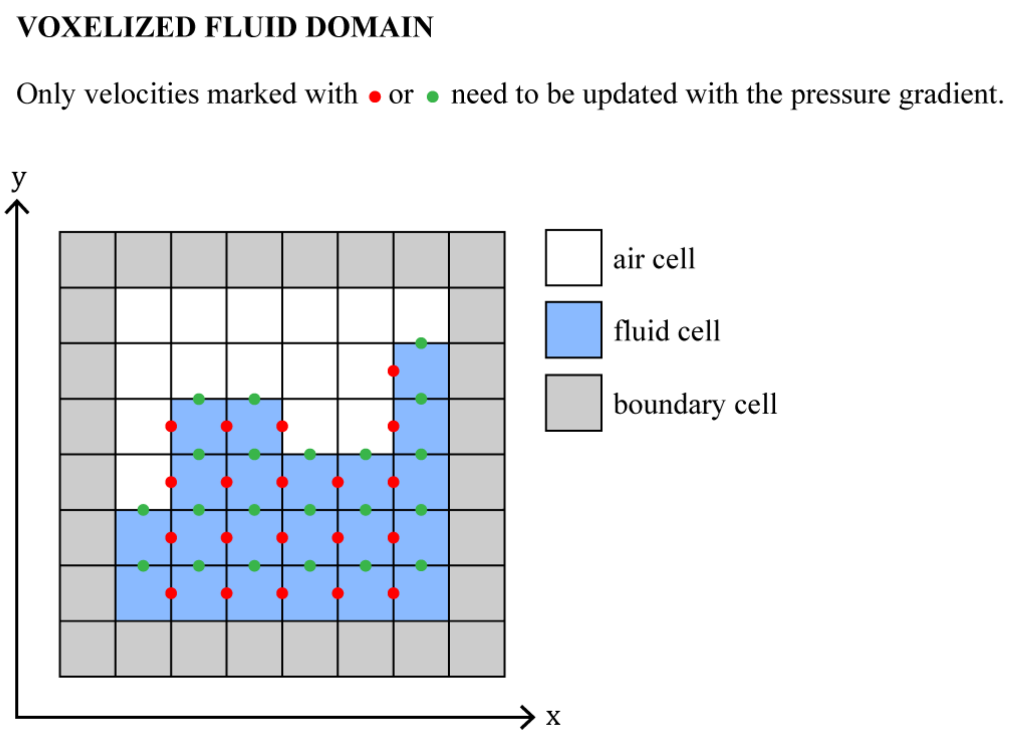

VOXELIZED FLUID DOMAIN
Only velocities marked with ● or ● need to be updated with the pressure gradient.
y
x
air cell
fluid cell
boundary cell

Figure 9 The voxelized fluid domain and velocities affected by the pressure gradient update

4.6 The discrete divergence

So far only the pressure update has been accounted for (equation 2.5), but there is still the incompressibility condition to be satisfied (equation 2.6). Luckily, given the way velocity is stored on the grid, computing the divergence is really straightforward.

The divergence in three dimensions is:

$$\nabla \cdot \vec{u} = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z}$$

and approximating it for cell (i, j, k) using central finite differences results in:

$$\nabla \cdot \vec{u}_{i,j,k} \approx \frac{u_{i+1/2,j,k} - u_{i-1/2,j,k}}{\Delta x} + \frac{v_{i,j+1/2,k} - v_{i,j-1/2,k}}{\Delta x} + \frac{w_{i,j,k+1/2} - w_{i,j,k-1/2}}{\Delta x} \quad (2.8)$$

As was the case with the pressure update, it is not required to calculate the divergence for the entire grid, but only for cells marked as fluid, because it is not important if air or solid objects change their volume or not.

4.7 The pressure equations

While the formula for how to update the velocity with the pressure gradient has been established, something is obviously missing: the pressure! The pressure update only describes *how* to update velocity, but gives no immediate clue on what the pressure values should be. Indeed, it has been previously established that pressures in the air are manually set to a constant value of zero (what is called a **Dirichlet** boundary condition). And while not discussed until now, the pressure value inside solids, as stated by the solid boundary condition, amounts to specifying the normal derivative of pressure $\partial p / \partial \hat{n}$ instead, rather than storing an explicit pressure value (what is called a **Neumann** boundary condition). But that is of no concern, since velocities at fluid-solid boundaries are manually set on each timestep anyway. So these two boundary conditions aside, all the pressures inside the fluid are still unknown!

So what has to be figured out before the fluid can come to life in all its incompressible glory is what values exactly does pressure need to have *inside* the fluid in order to achieve incompressibility when it updates velocity. And the way to do solve this problem is to take the two pieces of known information: how the pressure updates velocity and what condition the resulting velocity needs to satisfy and combine them together in a linear system of equations (one equation for each fluid cell) to solve for the unknown value. More specifically, equation (2.7) will be substituted into equation (2.8) like so:

$$\nabla \cdot \vec{u}_{i,j,k} \approx \frac{u_{i+1/2,j,k} - u_{i-1/2,j,k}}{\Delta x} + \frac{v_{i,j+1/2,k} - v_{i,j-1/2,k}}{\Delta x} + \frac{w_{i,j,k+1/2} - w_{i,j,k-1/2}}{\Delta x} = 0,$$

$$\begin{aligned}
&\frac{1}{\Delta x}\Bigg[\left(u_{i+1/2,j,k} - \Delta t\frac{1}{\rho}\frac{p_{i+1,j,k} - p_{i,j,k}}{\Delta x}\right) - \left(u_{i-1/2,j,k} - \Delta t\frac{1}{\rho}\frac{p_{i,j,k} - p_{i-1,j,k}}{\Delta x}\right) + \\
&\left(v_{i,j+1/2,k} - \Delta t\frac{1}{\rho}\frac{p_{i,j+1,k} - p_{i,j,k}}{\Delta x}\right) - \left(v_{i,j-1/2,k} - \Delta t\frac{1}{\rho}\frac{p_{i,j,k} - p_{i,j-1,k}}{\Delta x}\right) + \\
&\left(w_{i,j,k+1/2} - \Delta t\frac{1}{\rho}\frac{p_{i,j,k+1/2} - p_{i,j,k}}{\Delta x}\right) - \left(w_{i,j,k-1/2} - \Delta t\frac{1}{\rho}\frac{p_{i,j,k} - p_{i,j,k-1}}{\Delta x}\right)\Bigg] = 0
\end{aligned} \tag{2.9}$$

for a fluid cell (i, j, k). Algebraically simplifying the equation a bit yields what is a numerical approximation to the Poisson problem $-\Delta t / \rho \nabla \cdot \nabla p = -\nabla \cdot \vec{u}$:

$$\frac{\Delta t}{\rho}\left(\frac{\begin{array}{c}6p_{i,j,k}-p_{i+1,j,k}-p_{i,j+1,k}-p_{i,j,k+1}\\-p_{i-1,j,k}-p_{i,j-1,k}-p_{i,j,k-1}\end{array}}{\Delta x^2}\right)=-\left(\begin{array}{c}\frac{u_{i+1/2,j,k}-u_{i-1/2,j,k}}{\Delta x}+\\\frac{v_{i,j+1/2,k}-v_{i,j-1/2,k}}{\Delta x}+\\\frac{w_{i,j,k+1/2}-w_{i,j,k-1/2}}{\Delta x}\end{array}\right) \tag{2.10}$$

This is the general pressure equation formula for each FLUID cell in the grid. Note, however, that if the cell is at the fluid's boundary, there are already known values for pressure (as dictated by the boundary conditions) that need to be substituted into equation (2.9) in order to obtain a slightly different version of equation (2.10). Since differences would be minimal, they are simply going to be mentioned without rewriting the entire equation.

- For an air cell neighbour, the corresponding *p* is simply removed from the equation
- For a solid cell neighbour, the corresponding *p* is removed from the equation, but the integer coefficient in front of $p_{i,j,k}$ is also decreased by 1

4.8 Finding and applying pressure

After equation (2.10) is established for every fluid cell in the grid, what results is a large system of linear equations that needs to be solved for the unknown variables: the pressures *p*. The system can be thought of a coefficient matrix, *A*, times a vector of all the unknown pressures x, equal to a vector of negative divergences for each fluid cell, *b*:

$$Ax = b \tag{2.11}$$

There are many different methods of solving linear systems, but analyzing them was not a priority for the project. Optimized linear system solvers are readily-available in many C++ math libraries (more info in chapter 5), therefore implementing one was not considered important. After the linear system solver returns the computed pressures for each cell, the fluid velocities can be updated according to equation (2.7) and the newly obtained velocity field $\vec{u}^{n+1}$ (which is divergence-free and abides by the boundary conditions) can be used to advect the marker particles as the very last step of the entire fluid calculation.

Before closing the chapter, an outline of the routine followed for the pressure update is listed below. In code, all the pressure update steps are concatenated into a single function, project( ):

- Calculate the negative divergence for each fluid cell (with modifications at solid boundaries)
- Build the matrix of coefficients A, by looping over each FLUID cell and finding out what type of cells (fluid, air or solid) are neighbouring them
- Solve the linear system *Ax=b* using a linear system solver, which returns the vector of solved pressures *x*
- Compute the new velocity field $\vec{u}^{n+1}$ from $\vec{u}$ and the vector *x* using the pressure gradient update formula (equation (2.7))

5 Implementation

Chapter 4 described the majority of algorithms and formulas used to create the fluid simulator and for the most part, they are not going to be dissected again from a programming standpoint, unless there is a notable or important difference between theory and implementation. Instead, this chapter will mainly focus on describing the high-level relations between the classes and how they work together.

5.1 Software architecture and class analysis

The class structure is very simple and does not shy away from favouring "generalist" classes over very specialized ones. An approach where different sets of tasks are delegated to many smaller, specialized classes can be extremely powerful, but then again, so can having larger classes capable of managing many different types of information. Either way, the primary goal was to have classes that made logical sense.

The container class of the program (not detailed in the class diagram) is an **NGLScene** class inheriting from **QOpenFLWindow**. Its responsibilities are related almost exclusively to OpenGL. It is the class through which VBOs and VAOs are created and through which all OpenGL draw calls are made. Instead of having multiple classes each implement their version of draw(), all the drawing is handled by NGLScene instead. What data goes in the draw() function then is a generic geometry object defined by the **ShapeData** class. ShapeData can create indexed or non-indexed vertex arrays to be sent to OpenGL. It's really not more complex than that, as it does not do any type of checking on the generated data, but simply passes it down to draw(). All the classes that need to output something to be drawn on the screen output ShapeData objects, from the grid mesh to the particles. There is also a **ShapeFactory** class that can output various types of primitive ShapeData objects (cell, sphere, plane), which can come quite in handy.

Moving on to the actual simulator classes, a decision was made early on to have a **Grid** superclass that would act like a control hub for all the other classes. After all, most of the fluid information is stored on the grid, so it made sense to bypass any "manager" classes and simply provide the grid with direct control on the grounds that "if it belongs to the grid, then the grid should handle it". In turn, some of the classes that are connected to the grid usually store a pointer to it through which they can query information about any other connected classes.

First and foremost, the grid stores meta information about itself: dimensions in x, y and z, number of cells, where the boundaries are. Then, it stores an array of cells that make up the actual grid (what is meant through "array" is actually a std::vector, but to avoid any terminology confusion, a convention is made to substitute the first term for the latter for the remainder of the chapter). It also stores a lot of information about the fluid: density, slip condition, but most importantly, all the velocity and pressure values required in the fluid calculation. For intuitiveness purposes, it's usually cells that these values are retrieved through (associating a velocity with a cell is much more intuitive that indexing directly into a long array), but the storage happens on the grid, not on the cells. One advantage is that data is centralized and that memory is contiguous this way, but no tests were made to check whether it affects performance in any way. Time variables are stored on the grid as well: the id of the timer that controls the OpenGL window update, the current time, the timestep and the frame duration. The timestep, for instance is referenced by many other objects through the grid pointer they store. Other things stored on the grid are an integrator to perform all the integration tasks and the special objects used in solving linear systems of equations (required in the pressure projection step): a sparse matrix, a dynamic vector and a conjugate gradient solver. All objects come from the Eigen math library.

As for functionality, Grid does quite a few things, the most important being:

- Manages the cells it contains. Managing the cells is made easy by a series of handy accessor functions that allow for multiple ways of reaching them: by index, by spatial position or by grid coordinates.
- Executes all the steps of the simulation, from advection to velocity interpolation to pressure projection. Even though some of the sub-tasks are performed by other objects the grid keeps track of everything
- Creates geometry data to pass to OpenGL (cells, velocity fields, arbitrary vectors and other such data needed especially in the visual debugging)
- Generally facilitates information exchange between objects connected to it.

The **Cell** class is much simpler than Grid, though judging by its size in the UML diagram, one could think otherwise. Cell has a lot of getters and setters for its pressure and its three velocity components (plus their duplicates, used in various places –e.g in the backwards particle trace during the advection step), as well as accessors for its cell neighbours. An important thing to point out is that the cell does not store neither velocity nor pressure on itself, but rather keeps pointers into the arrays stored on the grid. In other words, a cell is only aware of its position in space and within the grid and does not actually hold the values it is associated with.

Emitter and **Particle** classes go hand in hand, obviously. They are rather simple classes and their main duty is of course to deal with the marker particles. The emitter can extra add particles into the scene and also integrates the particles in time when instructed to do so by the grid. Quite a few methods and member variables on these two classes ended up not actually being used, but are there for possible extensions in the future.

Finally, the last class that is of interest is the **Integrator**. This is a micro-class that has only two methods, the most important of which, *integrate()*, is a pure virtual method. Integrator is thus a parent class to four types of integrators used in the numerical simulation of the fluid: **Euler**, **RK2**, **RK3** and **RK4**. Having an abstract parent class makes it easy to store a pointer to an Integrator object without having to worry what type of integrator it is and also makes for easier changes in the code when switching between integrator types. Two classes store pointers to an Integrator object: Grid and Emitter, since they both need to integrate various quantities.

Overall, as has been previously said, the inner wirings between classes are quite simple and they (hopefully) make for a not too difficult understanding of how data flows in the program.

5.2 Programming challenges

One of the more difficult problems to solve was that of trilinear interpolation, not because of its core algorithm, but because of the staggered arrangement of the velocity. Three interpolations need to be made in order to get the full velocity vector and because velocity components are stored in different places both spatially and programmatically (arrays with different indices), it means that interpolating staggered values on a grid can be quite a pain, as acknowledged by Cline[2013] and even by Bridson [2015].

Another somewhat difficult problem that is worth mentioning was in the pressure calculation step, when calculating pressure coefficients for the linear system, because the matrix that needed to be filled with values was sparse. Sparse matrices require a lot of care, because it is very easy to put something in the wrong place. Using a debugging function to print out the values in an easy to read manner would be a good thing to have and could save a lot of headaches.

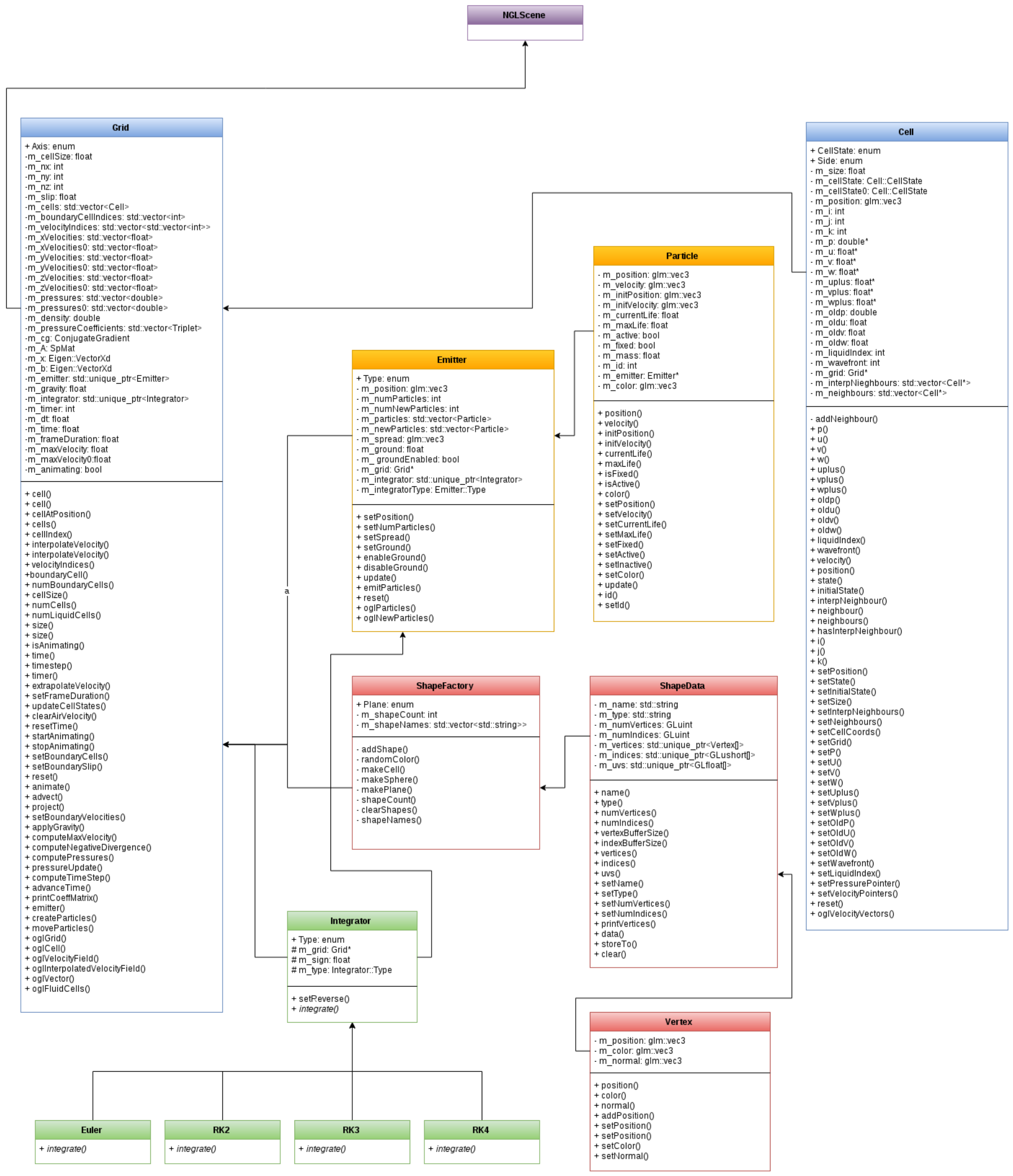
Finally, the velocity extrapolation function can very easily bring unwanted bugs (sometimes hard to trace) if not performed in the right way. Again, using a debugging tool (visual if possible) is recommended to make sure values are correct and to identify problem areas.

5.3 Other implementation details

The project makes use of classes from four external libraries: NGL, glm, Eigen, and OpenMP. Usage of NGL is limited to the use of a container class, NGLScene. Both glm and Eigen are only used for only a handful of classes (vec3, respectively SparseMatrix, ConjugateGradient and VectorXd), and OpenMP is used to parallelize bits of the code that are suited to parallelization. Most of the rest of the code is based on equations and pseudocode from the refrenced papers and books. Meaningful comments and overall code tidiness were enforced as much as possible, to facilitate work on the project later on.

5.4 UML diagram

Nothing important needs to be added here, apart from what has already been written in the previous section. The UML diagram omits a few details to keep things as clear and as simple as possible. Colour-coding is used to group together classes within the same "family".



6 Results

6.1 Functionality and performance

The simulation is basic. Liquid volume can be initialized at the start of the simulation by means of creating a smaller or larger cluster of particles at the position of the emitter. The simulation is then let to run on the fluid, creating fluid motion inside the grid container. Additional particles can also be created during the simulation by pressing down a hotkey.

A good feature of the program represents the ability to visualize velocities and other elements of the grid simulation. Things like active fluid cells, average cell velocity, the interpolated velocity field at arbitrary resolution and velocity components per cell face can be toggled on and off. The benefits of having such tools when debugging are worth the time invested into them.

Performance is acceptable, with low-res simulations running at high enough speeds to be classed as interactive. During the limited testing, particle numbers as high as ~100000 have been used, with the expected big slowdowns, however the biggest bottleneck is by far the grid resolution. Efforts have been made to parallelize code wherever possible and while that does visibly speed up the program execution, any notable advantage is negated by using a high resolution grid.

6.2 Limitations and future work

As is immediately evident from running the program, it has succeeded only in part. Not enough testing has been done to check whether that is partly to blame on the coarseness of the grid (only very low grids of maximum 12x12 cells have been used in testing) or whether it is due actual errors in implementing the algorithms. In any case, with the current settings the fluid does not behave like a believable liquid. There is a copious amount of velocity dissipation, especially around the borders, thus the fluid, even though it does not use the viscosity term and should technically be inviscid, behaves more like a viscous fluid. A weird displacement effect can also be noticed at the surface of the fluid, with particles manifesting (again) a viscous motion that prevents them to flow back into the body of the fluid. Researching the problem in the available academic literature hinted at the possibility of this being a side effect of sub-cell motion (movement that is less than one cell high), which seems plausible, since the fluid settles down within the height of only two grid cells.

Another encountered issue was the rapid decrease in performance as particles are being added into the system during the simulation, but so far the exact reason for why this is happening has not been determined.

7 Conclusion

This was only a partially successful project and it fell short even on the modest goals it set at the beginning. That being said, valuable lessons have been learned and improving and extending the program are definite goals for the future.

Bibliography

BRACKBILL, J.U. AND RUPPEL, H.M. 1986. FLIP: A method for adaptively zoned, particle-in-cell calculations of fluid flows in two dimensions. In *Journal of Computational Physics*, 65(2), 314-343.

BRIDSON, R. 2015. *Fluid Simulation for Computer Graphics*. CRC Press.

CLINE, D., CARDON, D. AND EGBERT, P.K. 2013. *Fluid flow for the rest of us: Tutorial of the marker and cell method in computer graphics.*

ENRIGHT, D., FEDKIW, R., FERZIGER, J. AND MITCHELL, I. 2002. A hybrid particle level set method for improved interface capturing. In *Journal of Computational physics*, 183(1), 83-116.

EVANS, M.W., HARLOW, F.H. AND BROMBERG, E. 1957. *The particle-in-cell method for hydrodynamic calculations* (No. LA-2139). LOS ALAMOS NATIONAL LAB NM.

FERSTL, F., ANDO, R., WOJTAN, C., WESTERMANN, R. AND THUEREY, N. 2016. Narrow band FLIP for liquid simulations. In *Computer Graphics Forum*, Vol. 35, No. 2, 225-232.

FOSTER, N. AND FEDKIW, R. 2001. Practical animation of liquids. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, ACM, 23-30.

FOSTER, N. AND METAXAS, D. 1996. Realistic animation of liquids. In *Graphical models and image processing*, 58(5), 471-483.

GINGOLD A. AND MONAGHAN J.J. 1977. Smoothed particle hydrodynamics-Theory and application to non-spherical stars, In *Mon. Not. Roy. Astron. Soc.*, vol. 181, 375–389.

GOURLAY, M.J., Fluid simulation for video games (part 1).
https://software.intel.com/en-us/articles/fluid-simulation-for-video-games-part-1.
Accessed: 2016-08-19.

HARLOW, F.H. AND WELCH, J.E. 1965. Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface. In *Physics of fluids*, 8(12), 2182.

HESS, J.L. AND SMITH, A.O. 1967. Calculation of potential flow about arbitrary bodies. In *Progress in Aerospace Sciences*, 8, 1-138.
MÜLLER, M., CHARYPAR, D. AND GROSS, M. 2003. Particle-based fluid simulation for interactive applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, Eurographics Association, 154-159.

OSHER, S. AND FEDKIW, R.P. 2001. Level set methods: an overview and some recent results. In *Journal of Computational physics*, 169(2), 463-50.

PARENT, R. 2012. *Computer Animation: Algorithms and Techniques*. Morgan Kaufmann.

SIDEFX SOFTWARE. Flip solver.
http://archive.sidefx.com/docs/houdini15.5/nodes/dop/flipsolver.
Accessed: 2016-08-19.

STAM, J. 1999. Stable fluids. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques,* ACM Press/Addison-Wesley Publishing Co., 121-128.

STAM, J. AND FIUME, E. 1993. Turbulent wind fields for gaseous phenomena. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, ACM, 369-376.

STAM, J. AND FIUME, E. 1995. Depicting fire and other gaseous phenomena using diffusion processes. In *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '95, New York, NY, USA, 129–136

ZHU, Y. AND BRIDSON, R. 2005. Animating sand as a fluid. In *ACM Transactions on Graphics*, Vol. 24, No. 3, ACM, 965-972.